

datatype in the static diagram to create a datatype descriptor. The packages handler handles each package in the static diagram to create a package descriptor. The interfaces handler handles each interface in the static diagram to create an interface descriptor. The models handler handles each model in the static diagram to create a model descriptor. The signals handler handles each signal in the static diagram to create a signal descriptor. The associations handler handles each association in the static diagram to create a model descriptor which is a process entity for that association.

[0463] In FIGS. 10 and 11, the method to convert model state charts to model state enumerations and model operations is described. Each model state chart (Model<n> State Chart) may contain zero-to-many States (State<1>, State<2>, . . . , State<n>). Events trigger the change between one state and the next state (as indicated by "Action/Event" on the association connectors). When an event occurs triggering a state change one or more actions may be executed (as indicated by "Action/Event" on the association connectors). An action may be marked as an entry action for a state or as an exit action for a state. Actions may also contain zero-to-many subactions. A Model State Chart describes the allowable states into which a model can enter, the actions (operations) associated with entering that state, and the actions (operations) associated with exiting from that state.

[0464] In the Model<n> State Chart, events cause a state attribute to move between an Initial State, State<1>, State<2>, State<3>, State<n>, and a Terminal State. The Model<n> State Chart is read by a State Chart Handler that includes a States Handler, a Signals Handler, and an Actions Handler. As indicated by connector B in FIGS. 10 and 11, the States Handler generates an enumeration implementation with an enumeration value for each state. As indicated by connector C in FIGS. 10 and 11, the Signals Handler generates a signal descriptor for each signal. As indicated by connector D in FIGS. 10 and 11, the Signals Handler generates a signal method descriptor for each signal. As indicated by connector E of FIGS. 10 and 11, the Actions Handler generates an Enter State Method Descriptor for each entry action. As indicated by connector F of FIGS. 10 and 11, the Actions Handler generates an Exit State Method Descriptor for each state exit action. The Model State Enumeration and the Signals Descriptors are stored in the Metamodel repository of the Component Integration Engine. The Signal Method Descriptor, the Enter State Method Descriptor, and the Exit State Method Descriptor are each added to a Model Descriptor that, in turn, is stored in the Metamodel Repository.

[0465] In FIG. 12 activity diagrams create method descriptors or flowchart assemblies. Some activity diagrams show the activity (operations) performed by a method (an operation which is a feature on a model). These activity diagrams are converted to Method descriptors and set on model descriptors as a method descriptor. Some activity diagrams show activity that is not specific to one model and are transformed into flowchart assemblies. Flowchart assemblies are simply component assemblies designed to perform a specific operation. Flowchart assemblies may use language construct operations such as: loops, branches, variables, etc. or may call static methods on models or instance methods on instances. As discussed previously, component assemblies are similar to models but have a shorter duration of use.

Flowchart assemblies are given a name and saved in the component integration engine as a component assembly.

[0466] A Model Method Descriptor or Flowchart Assembly of the Component Integration Engine includes a Sub-operation Descriptor including zero-to-many virtual operations (Virtual Operation<1>, a Virtual Operation<2>, an If/Else Operation, and Loop Operation). Some operations may control which other operations are executed (If/Else Operation) and others may control how many times other operations are executed (Loop Operation). As shown in FIG. 12, a Method<1> Activity Diagram includes a Process<1> that transfers control to a Process<2> when Process<1> terminates. If the If Condition of the If/Else Operation is satisfied, then Process<2> transfers control to Process<3> else Process<2> transfers control to Process<4>. After Process<3>, terminates, Process<3> transfers control to Process<n> After Process<4> terminates, Process<4> transfers control to Process<n>. Process<n> then loops upon itself until the Loop Condition of the Loop Operation is satisfied. Process<1> maps to Virtual Operation<1>. Process<2> maps to Virtual Operation<2>. The If/Else decision diamond between Process<3> and Process<4> maps to the If/Else Operation. Process<3> Maps to the If Statement of the If/Else Operation. Process<4> maps to the Else Statement of the If/Else operation. The decision diamond for the loop for Process<n> maps to the Loop Statement.

[0467] FIG. 13 illustrates some of the enumerations pre-defined in the metamodel repository of the component integration engine due to the enumerations' use of UML. New enumerations may be created and added just as with any other modeling element.

[0468] As shown in FIG. 13, an Enumeration Implementation creates an Enumeration Instance. The Enumeration Implementation includes the Attribute Implementations: Name, Description, Type and Values. The Enumeration Instance includes a Name, a Description, a Type and one or more values (indicated by Value<1>, Value<1>, and Value<n>).

[0469] Also as shown in FIG. 13, are the Metamodel Enumeration Instances of the Component Integration Engine: Call Concurrency, Changeability, Model Stereotype, Constraint Stereotype, Constraint Kind, Language, Multiplicity, Operation Kind, Parameter Kind, Scope, Visibility. The Metamodel Enumeration Instances are pre-defined by the Metamodel Repository. The metamodel repository allows for the definition of new enumerations, but predefines those enumerations related to the metamodeling process.

[0470] In FIG. 14, a user defines a Metahint Descriptor that includes a Name, a Description and Attribute Descriptors. This metahint descriptor is saved to the metamodel repository (specifically in the meta-metarepository section of the metamodel repository). As indicated by connector G of FIGS. 14 and 15, the Metahint Descriptor is read by a Metahint Impl(ementation) Generator, creates a Metahint Implementation, and creates a Hint Descriptor of FIG. 15. Instances of a metahint implementation are hint descriptors that describe the model used to describe a hint.

[0471] As indicated by connector H of FIGS. 14 and 15, the Attribute Descriptors are read by an Attribute Handler of